图文 73 Consumer消息零丢失方案：手动提交offset + 自动故障转移

详情 评论

Consumer消息零丢失方案：手动提交offset + 自动故障转移

继《从零开始带你成为JVM实战高手》后，救火队长携新作再度出山，重磅推荐：

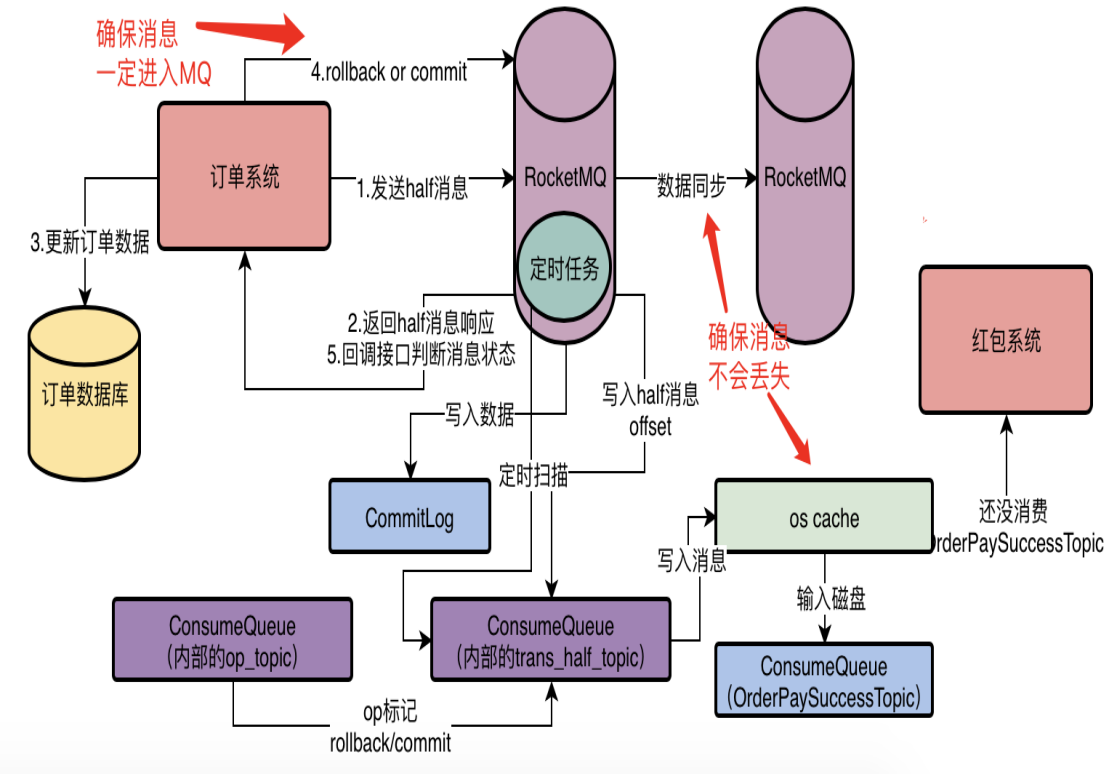
（点击下方蓝字试听）

《从零开始带你成为MySQL实战优化高手》

1、红包系统拿到了消息就一定会派发红包吗？

通过前面的学习，我们现在已经知道了如何确保订单系统发送出去的消息一定会到达MQ中，而且也能确保了如果消息到达了MQ如何确保一定不会丢失

我们看下面的图示意了这个场景

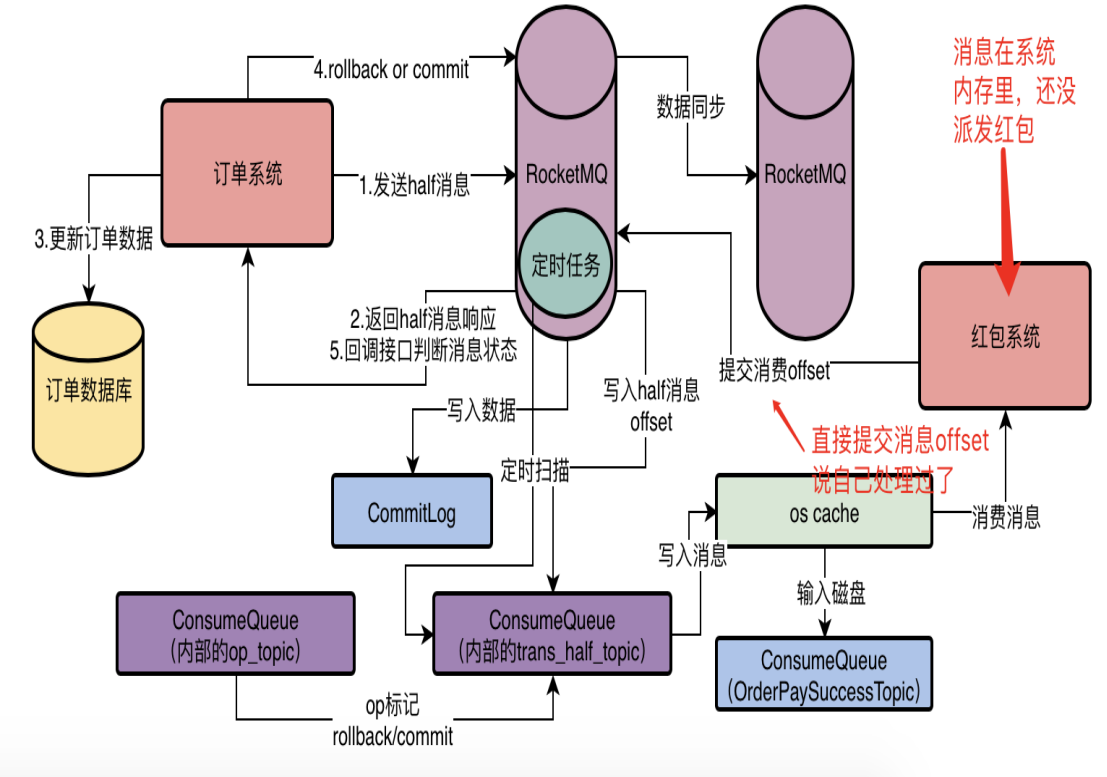


只要我们能做到这一点，那么我们必然可以保证红包系统可以获取到一条订单支付成功的消息，然后一定可以去尝试把红包派发出去。

但是现在的问题在于，即使红包系统拿到了这条消息，就一定可以成功的派发红包吗?

答案是未必

我们之前也给大家分析过这个问题，如果红包系统已经拿到了这条消息，但是消息目前还在他的内存里，还没执行派发红包的逻辑，此时他就直接提交了这条消息的offset到broker去说自己已经处理过了，我们看下图。



接着红包系统在上图这个状态的时候就直接崩溃了，内存里的消息就没了，红包也没派发出去，结果Broker已经收到他提交的消息offset了，还以为他已经处理完这条消息了。

等红包系统重启的时候，就不会再次消费这条消息了。

关于这个问题，我们之前通过画图的方式，已经清晰的展示了消息offset错误提交之后，导致红包系统可能无法再次获取到这条消息的问题。

所以我们在这里，首先要明确一点，那就是即使你保证发送消息到MQ的时候绝对不会丢失，而且MQ收到消息之后一定不会把消息搞丢失，但是你的红包系统在获取到消息之后还是可能会搞丢。

2、Kafka消费者的数据丢失问题

虽然我们这个专栏主要是依托RocketMQ来讲解消息中间件技术的原理、生产架构以及技术方案的，但是其实我们这里涉及到的各种技术思想，包括MQ数据丢失问题以及解决方案，在Kafka、RabbitMQ等其他中间件里也是完全适用的。

所以对我们目前讲解的这个消费者数据丢失的问题，其实完全可以套用到Kafka中去，因为Kafka的消费者采用的消费的方式跟RocketMQ是有些不一样的，如果按照Kafka的消费模式，就是会产生数据丢失的风险。

也就是说Kafka消费者可能会出现上图说的，拿到一批消息，还没来得及处理呢，结果就提交offset到broker去了，完了消费者系统就挂掉了，这批消息就再也没机会处理了，因为他重启之后已经不会再次获取提交过offset的消息了。

3、RocketMQ消费者的与众不同的地方

但是对于RocketMQ的消费者而言，他是有一些与众不同的地方的，至少跟Kafka的消费者是有较大不同的

我们再来回顾一下之前我们写过的RocketMQ消费者的代码，如下所示。

```
public class RocketMQConsumer {

    public static void start() {

        new Thread() {

            public void run() {

                // 这是RocketMQ消费者实例对象
                // “credit_group”之类的就是消费者分组
                // 一般来说比如积分系统就用“credis_consumer_group”
                // 比如营销系统就用“marketing_consumer_group”
                // 以此类推，不同的系统给自己取不同的消费组名字
                DefaultMQPushConsumer consumer = new   DefaultMQPushConsumer("credit_group");

                // 这是给消费者设置NameServer的地址
                // 这样就可以拉取到路由信息，知道Topic的数据在哪些broker上
                // 然后可以从对应的broker上拉取数据
                consumer.setNamesrvAddr("localhost:9876");

                // 选择订阅“TopicOrderPaySuccess”的消息
                // 这样会从这个Topic的broker机器上拉取订单消息过来
                consumer.subscribe("TopicOrderPaySuccess");

                // 注册消息监听器来处理拉取到的订单消息
                // 如果consumer拉取到了订单消息，就会回调这个方法教给你处理
                consumer.registerMessageListener(new MessageListenerConcurrently() {
                    @Override
                    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                                                                    ConsumeConcurrentlyContext context) {
                        // 在这里对获取到的msgs订单消息进行处理
                        // 比如增加积分、发送优惠券、通知发货，等等
                        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                    }
                });

                // 启动消费者实例
                consumer.start();
                  System.out.printf("Consumer Started.%n");

                while(true) { // 别让线程退出，就让创建好的consumer不停消费数据
                    Thread.sleep(1000);
                }
            }
        }.start();
    }
}
```

我们再回顾一下这里的具体的一小块代码：

```
consumer.registerMessageListener(
        new MessageListenerConcurrently() {
                @Override
                public ConsumeConcurrentlyStatus consumeMessage(
                        List<MessageExt> msgs,
                        ConsumeConcurrentlyContext context) {
                    // 在这里对获取到的msgs订单消息进行处理
                    // 比如增加积分、发送优惠券、通知发货，等等
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                }
        }
);
```

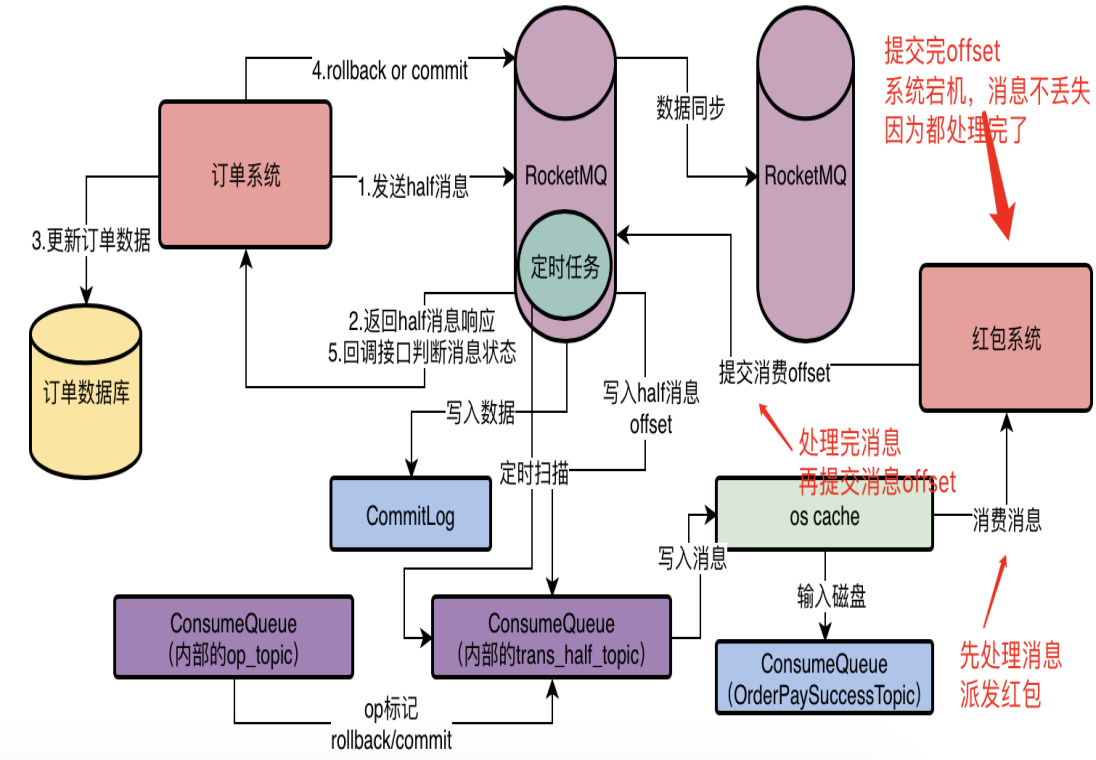
大家会发现，RocketMQ的消费者中会注册一个监听器，就是上面小块代码中的MessageListenerConcurrently这个东西，当你的消费者获取到一批消息之后，就会回调你的这个监听器函数，让你来处理这一批消息。

然后当你处理完毕之后，你才会返ConsumeConcurrentlyStatus.CONSUME_SUCCESS作为消费成功的示意，告诉RocketMQ，这批消息我已经处理完毕了。

所以对于RocketMQ而言，其实只要你的红包系统是在这个监听器的函数中先处理一批消息，基于这批消息都派发完了红包，然后返回了那个消费成功的状态，接着才会去提交这批消息的offset到broker去。

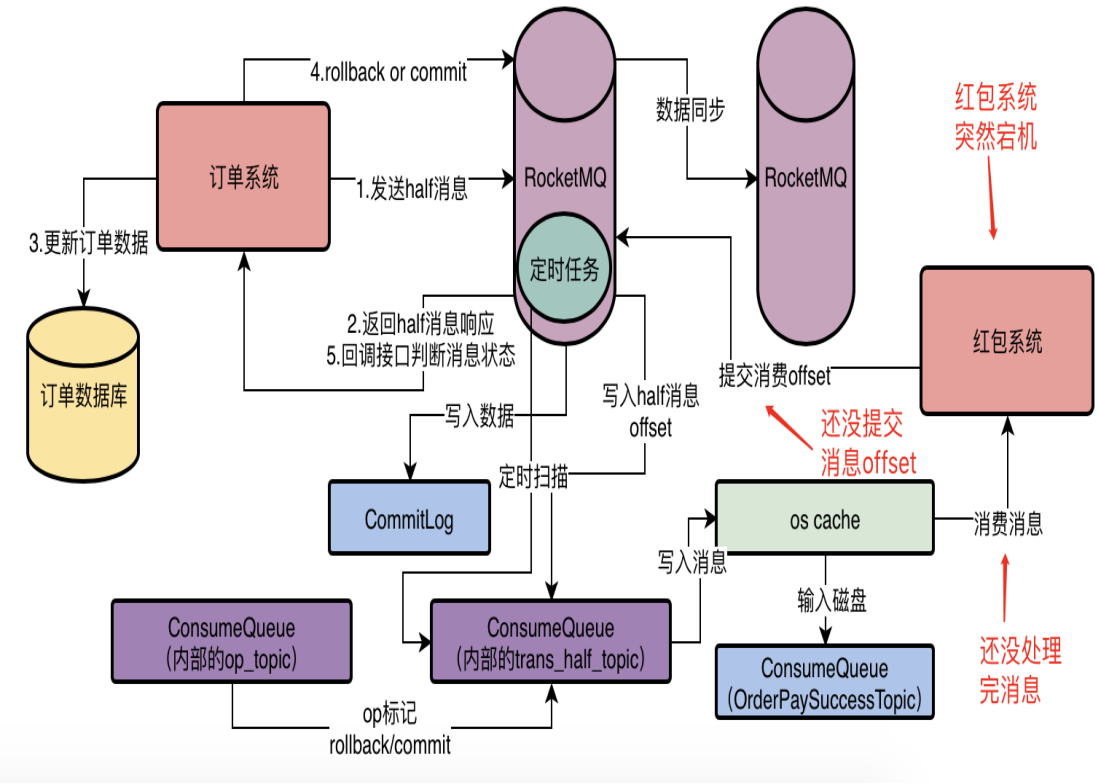
所以在这个情况下，如果你对一批消息都处理完毕了，然后再提交消息的offset给broker，接着红包系统崩溃了，此时是不会丢失消息的

我们看下图的示意



那么如果是红包系统获取到一批消息之后，还没处理完，也就没返回ConsumeConcurrentlyStatus.CONSUME_SUCCESS这个状态呢，自然没提交这批消息的offset给broker呢，此时红包系统突然挂了，会怎么样？

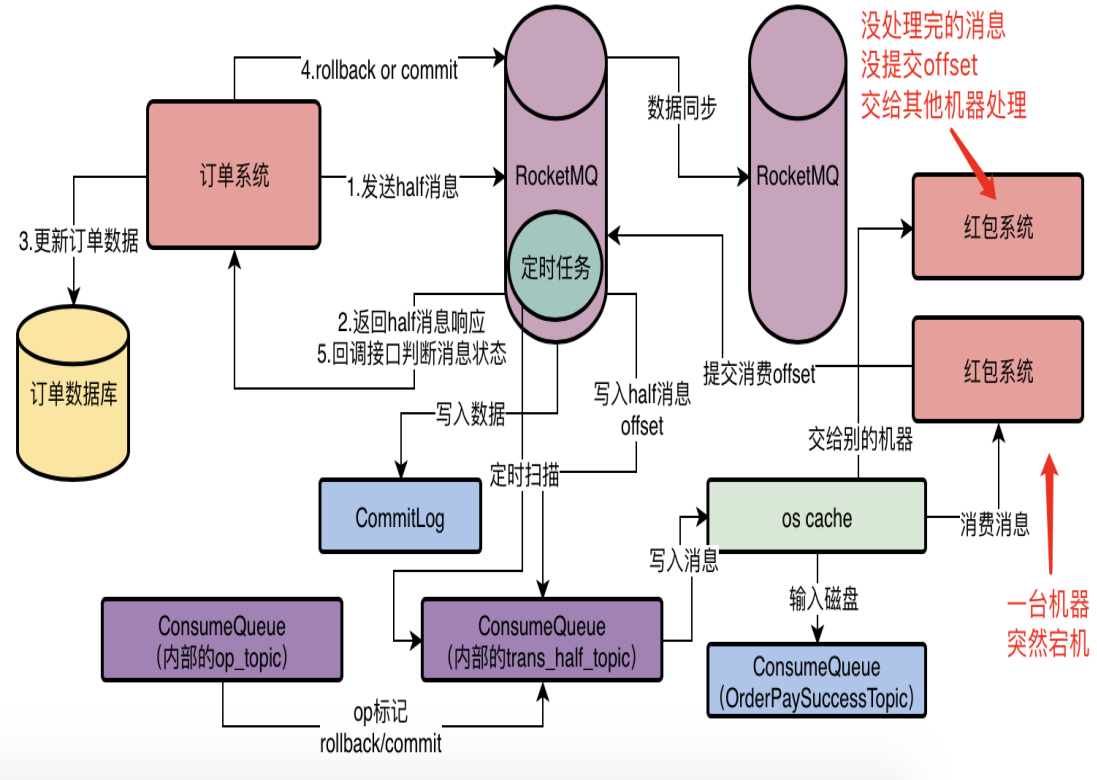
我们看下图示意的这个场景。



其实在这种情况下，你对一批消息都没提交他的offset给broker的话，broker不会认为你已经处理完了这批消息，此时你突然红包系统的一台机器宕机了，他其实会感知到你的红包系统的一台机器作为一个Consumer挂了。

接着他会把你没处理完的那批消息交给红包系统的其他机器去进行处理，所以在这种情况下，消息也绝对是不会丢失的

我们看下图的示意



4、需要警惕的地方：不能异步消费消息

所以大家也看到了，在默认的Consumer的消费模式之下，必须是你处理完一批消息了，才会返回ConsumeConcurrentlyStatus.CONSUME_SUCCESS这个状态标识消息都处理结束了，去提交offset到broker去。

在这种情况下，正常来说是不会丢失消息的，即使你一个Consumer宕机了，他会把你没处理完的消息交给其他Consumer去处理。

但是这里我们要警惕一点，就是**我们不能在代码中对消息进行异步的处理**，如下错误的示范，我们开启了一个子线程去处理这批消息，然后启动线程之后，就直接返回ConsumeConcurrentlyStatus.CONSUME_SUCCESS状态了

```
consumer.registerMessageListener(new MessageListenerConcurrently() {

        @Override
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeConcurrentlyContext context) {

            // 开启一个子线程处理这批消息
            new Thread() {
                public void run() {
                    // 处理消息
                }
            }.start();

            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
    }
);
```

如果要是用这种方式来处理消息的话，那可能就会出现你开启的子线程还没处理完消息呢，你已经返回ConsumeConcurrentlyStatus.CONSUME_SUCCESS状态了，就可能提交这批消息的offset给broker了，认为已经处理结束了。

然后此时你红包系统突然宕机，必然会导致你的消息丢失了！

因此在RocketMQ的场景下，我们如果要保证消费数据的时候别丢消息，你就老老实实的在回调函数里处理消息，处理完了你再返回ConsumeConcurrentlyStatus.CONSUME_SUCCESS状态表明你处理完毕了！

End

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：《付费用户如何加群》（**购买后可见**）